Universitat de Lleida

TREBALL FINAL DE GRAU

ESCOLA
POLITÈCNICA SUPERIOR
UNIVERSITAT DE LLEIDA
INSPIRING THE FUTURE

Estudiant: Ricard Plana Llauradó

Titulació: Grau en Enginyeria Informàtica

Títol de Treball Final de Grau: Design and implementation of a platform to integrate pharmacological data and grant access to perform studies and analytics and provide tools to exchang and disseminate the results.

Director/a: **Jordi Mateo Fornés, Dídac Florensa Cazorla**

Presentació

Mes: Juliol

Any: 2021

Contents

List of Figures

1 Introduction

The healthcare industry historically has generated large amounts of data, driven by record keeping, compliance and regulatory requirements, and patient care [1]. While most data is stored in hard copy form, the current trend is toward rapid digitalization of these large amounts of data.

This information requires a volume of requirements specifics to optimise the efficiency of the recollection and improve the quality of the analysis to add knowledge to the outcomes interpretation. Nowadays, one of the problems was the storage of this information and its analysis. Due to these big data sets, is required efficient tools that permit to build efficient analytic models.

A challenging problem in this domain is how this amount of data is used to benefit society and research to discover new patterns, build predict and prescriptive models, and more.

The main practical problem that confronts us is that nowadays, the used technologies to store the information is non-regulated or non-standardised relational databases storing yearly datasets. This situation is even more problematic because these databases are isolated and stored in personal computers or/and built with rudimentary or inefficient technology such as Acces [2] or Excel [3].

The main limitations of Acces databases are: that these databases are not available over the internet, having only 2 GigaByte(GB) file size limit and also unable to handle large quantities of database queries. Thus, it is mainly focused on small systems. Thus, making it way harder to integrate to more complex systems and devices in the healthcare sector.

Therefore, analysing and using this data requires enormous data-cleaning, scripting, and ETLs to prepare the context to feed analytical tools.

To overcome the challenges exposed until now, in this work we propose a generic architecture and to provide guidelines on how to adapt the architecture to different scenarios. Therefore the main contributions of this work are:

1. Propose data pre-processing strategy that maps data collection to a prepared environment transparent to the healthcare professional.
2. Design a scalable platform that permits build data-driven models on the top.
    - Designing a database to boost data queries.
    - Implementing an Application Programming Interface(API) to allow accessing the data.
    - Designing visualization tools to interact with the database.

In order to validate this architecture design this work presents a case study for the pharmacology sector.

2 Methodology

2.1 Architecture

We decided to divide our project in 3 different layers, with at least with a micro-service in it.

Data Layer will contain all the raw data and our database micro-system.

Logic Layer includes the different processes to import the data against the database and all the different scripts that are made to interact directly with the access files and the database, at the same time will contain the API and the different features it will provide to our platform.

Presentation Layer is focused mainly to contain the graphic interface we made to be able to check some data without the need to perform any request, and being able to check it in a way friendly view.

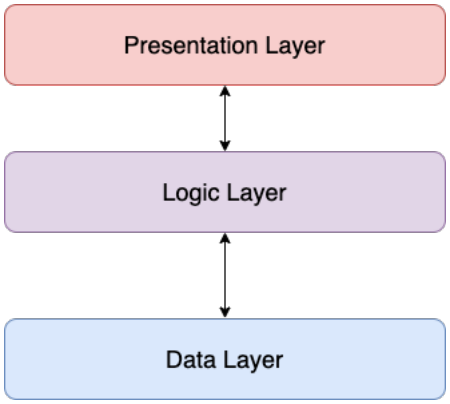


Figure 1: Layers Diagram

Nowadays there are a lot of different operating system, and we wanted to be sure that our project will work the same no matter the base it will be running.

To be able to accomplish that main idea, we used Virtualization, allowing us to create an abstraction layer over the hardware. At the same time, we wanted to be able to have a fast resource provisioning and speedier availability of new applications. So we decided to implement a Container Virtualization [4], containerizing the packages together everything needed to run a single application

or micro-services, including all the code, it's dependencies and even the operating system itself.

To be able to accomplish that, we used Docker [5], it is an open platform for developing, shipping and running applications, allowing as to separate our application from the infrastructure to deliver it quickly.

It made the fist part of the launch a bit harder, as we had to config our docker configuration before we were able to start to code or developed each part, being sure that it will properly work in the docker environment. Using docker, allowed us to be able to run all parts from our system (data integration, API and Interface) working as one. So the only thing we have to make when we want to start/reboot/stop some application will be done directly on the docker image. So if it needs to be deployed somewhere on the future, the only need is to move the project into the new environment and run the docker, and everything will start working.

While being the idea situation, it wasn't the easiest way, to be able to run everything from our docker it needed a previous work to make it work properly and sometimes it took so much time.
But thinking that we no longer have to worry about if it will work on a different environment than the development one, made the extra work worth it.

2.2 Data layer

Our first layer will contain all of the different Comma-Separated Values (csv) files exported from the Access platform and our non relational database. Working with non-relational database allows us to let our data not being confined to structural groups, performing functions that allows us for greater flexibility and making our data and analysis being more dynamic and allowing more variant inputs.

The project is based on MongoDB as a NoSQL database program to store our information in JSON-like documents with optional schemas.After being checking the different database the one that fulfilled most of our needs was MongoDB, it's a document-based, distributed database.

At the same time we focused on using MongoDB Compass, being the Graphical user interface (GUI) for MongoDB, allowing us to visually explore our data, run ad hoc queries in seconds, and view and optimize our query performance. Using the aggregation tool, we were able to arrange the API structure way faster thanks to the easy and fast help provided by the tool.

The main structure is focused on holding a main collection called treatments and some auxiliary document to make the interactions with specified collections easier and faster.

While deciding the different structure that the database will take, we had to change the indexes of the different collections several times. Mostly of them were done to be able to perform the different queries and matches from the API as fast as possible. While using the wrong indexes, some petitions from the API took several minutes, but we were able to reduce it, to less than a minute. In the end, when we have our Interface using different request at once, that different ends being a great improvement.

2.3 UML Diagram

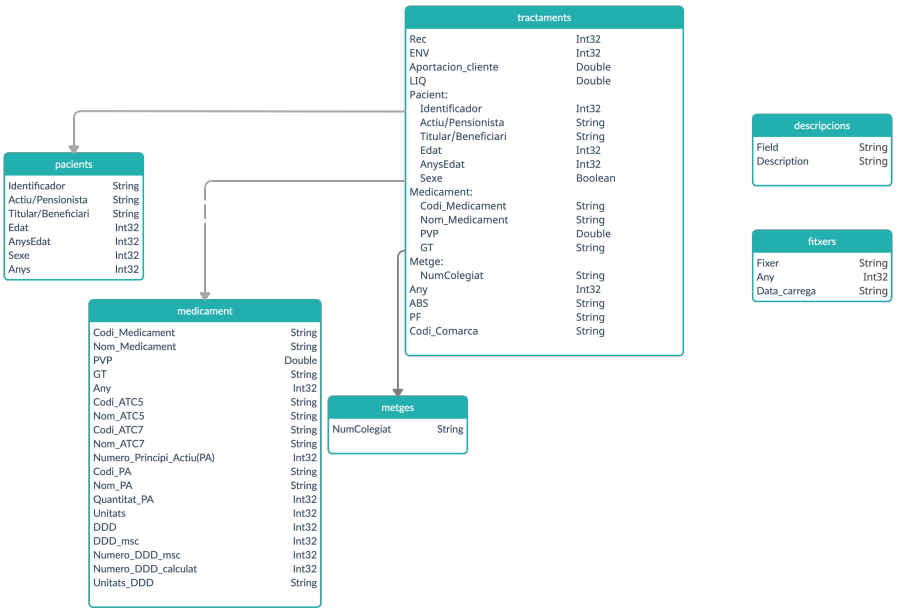


Figure 2: UML Diagram

2.4 Indexes

2.4.1 *Medicament*

That collection has a lot of different fields. Almost all of them are pure related to the healthcare field. But if we really think about how many of them we will be performing daily and in great quantity, we can just discard all of them but two. In the end we decided to focus in only use 2 indexes.

Codi_Medicament: It's one of the primary fields around all of our DataBase (DB), almost all of our access to the collection will be done using this field as the main one, no matter if it's just to retrieve a great quantity of data or just

sort a few. Being the most used field to perform that task were more than enough to be the primary index of our collection making it unique.

GT (*Grup Terapèutic*) : Another field that we decided to use as index, in this case it was promoted to index mainly because all the request related always involved to fetch a big quantity of data, so being able improve the performance of them was the reason config it as index.

2.4.2 *Pacients*

While working with the patients data, the easiest way to create a unique id to avoid duplicity was just using the textitidentificador as index, being that fields as the NIF for the differents patients around of system was the easiest way to have them tracked down. And at the same time was really a good point to improve performance due almost of the requests always were related to sort, group or count using that field as reference.

2.4.3 *Tractaments*

Tractaments is the main collection in our database and at the same time the biggest one with difference, using only data from one year we had around 6 millions of registers. At the same time, it's not only the biggest one, but the most used from our API. So deciding the indexes was a critic point. In the end we decided to use two indexes one for each index from a foreign collection being:

Medicament.Codi_Medicament: In this case when we are trying to perform requests trying to match all the info, we had available with some meds. We had to set it as index. While working with that high number of documents, we needed to be able to improve the performance as much as possible. And a great number of API services ended up being related to perform different actions using that field as reference.

Pacient.Identificador: In the end this was the same case as before, being one of the foreign indexes ended up to be used in different services and we ended up in the same, trying to perform a match, sort, group or other, took too much time. So setting it as index was a must, allowing to reduce the time and avoiding possible timeouts against the services.

2.5 Logic Layer

This layer will focus on all the items that will directly perform actions against objects of our system, including all the different scripts used during the importation and management of the data to the MongoDB and the ones used to perform maintain against some collections in it. At the same time, our API and all the different services that it provides to our system will also be included in it.

2.5.1 Database Import Implementation

First part of the project was to use the raw data and import it to our DB. All of our data was divided in two types of files, one for each year of Treatments and at the same time one unique file that stored all the information of the meds that are related in the Treaments and more that maybe they don't have registers, at least yeat.

So we developed two programs based on python called:

import_mongo.py thats the main one, being the one that will import all the data from the files. When working with that massive number of registers, we had to take a lot in consideration, to avoid unexpected behaviours or errors.

When the first version was ready, we realized that the performance wasn't as good as we were expecting, while working with test files it was working well, but at the moment we moved to use the real ones it wasn't practical. It needed of interaction to properly work, and at the same time it had a poor performance, dealing almost 5 hours to finish. While searching what could be the handicap in performance, we realized that the insert was doing one at time, making a lot of access to the DB and delaying all the process.

So for the version we decided as long as some improvements to modify it, to do it as bulk, instead of one by one. But we had to take care of the memory usage, we weren't able to bulk them at once at risk of overload the memory and crash the Server (SV). And the limitations of some functions used on the import of the data agains the mongo, set the max of registers to include at once at 100.000 [6]. So we decided to low it to 50.000 each time. As we are always trying to make our programs as friendly as possible to further modifications, we implemented that as a global var, so everyone can just modify it to adjust as max as their system allow them.

import_drugs.py was a little script in python, made to be able to add some meds and properties to the existend meds. We decided to implement as a new program instead to join it into the previous one, avoiding the increase the time treatments file would need to import, as this file of drugs it's though to only need to be imported once, without being related with the differents treatments files we would load in the future.

At the same time all the names related to fields and tables, are not hardcoded, they will be read from a file called *database_fields.csv* located with the import programs. And will be read at the moment to initialize the DB import, so in case there is a need to rename them in another language, they just need to be renamed and the import code will work without troubles.

2.5.2 API Implementation

While working with the API, we were trying to config, not only the different nodes needed for the project himself, but allowing the more generic request against our main collections. So if anyone that have access to our system needs to access some of the data there wouldn't be any need to modify the service itself, at least in the most common requests.

At the same time as soon as we had the main services running we started to develop the interface, the one that will be using our request and database to make the raw data, way easier to review it. In the start we were just getting the raw data from a request and send it to the interface, but at the same time it made the calcs in the interface way slower, as R wasn't the most efficient one while working with a great number of registers at once.

So almost all the main static request, that would not change we decided to move all of those operations directly on the DB, so we added some extra nodes that will allow us to directly fetch the data with the major calcs already done, some of them are, for example, group by, sort or count. So in the end in the interface, we only had to work directly with the data without need to add intermediate data management.

All the nodes are designed the same way trying to make them as generic as possible, so if we need to add one in the future, the only need will be to create one following the pattern and using the tool we described before the MongoDB Compass, it has an amazing option called aggregation, with this one we will be able to generate the body of our node, allowing us to be able to introduce new nodes on demand fast and easier with only a few minutes we could have an specified new node working.

2.5.3 Technologies

Visual Studio Code, by Microsoft, was the IDE majorly used in this project for the data integration. And allowed us to work directly using SSH connection to main server at some points.

In addition, Postman had been used as a support tool when building the different web services and for testing afterwards. It offers us an easy-to-use interface, with which to make HTML requests, without the hassle of writing a bunch of code just to test an API's functionality.

Finally, the API server side had been developed in Node an asynchronous event-driven JavaScript runtime, Node.js is designed to build scalable network applications.

Figure 3: API and Database technologies. VisualStudio Code, Postman, MongoDB, Node, respectively

2.6 Presentation Layer

When we finished or at least had the first serviced already working, the point with the highest impact in the project, was the way to show the results and how data will be displayed but it wasn't just that, right now in the marked there are a lot of differents languages that will allow us to setup the main interface.

But while deciding which one will fulfill in the best way our needs, we realized that a lot of people in the healthcare field were kinda related with R, so we wanted to give our apple that bit of freedom, so if anyone would like to adapt it to their needs or just modify it slightly we wanted to make that process.

In the end our choice was using R and to allow us to setup our interface we used Shiny from RStudio. An R package that makes it easy to build interactive web apps straight from R.
As the project was already using R and Shiny by RStudio, we decided to use the RStudio IDE, one already made to work with the architecture we had.

Figure 4: R, Shiny, RStudio, respectively

2.6.1 Interface Implementation

In the implementation of the interface, we decided to make it entirely in R and RShiny as we explained before it's easier if someone of the related field needs to directly modify it, or needs to create a new custom one, using our own as template.

While developing it, we realized that being able to perform the request directly on mongo, improved the performance.

So when we had to think about how to improve the performance, we focused on making the changing fields dynamics, allowing us to hold some of the fields static, so the only time they will fetch will be the first time we load the page, allowing us to make the next interaction way faster.
So we moved all the static data, into the header of the page, making it the first thing would be prompt directly to the user.

Figure 5: Interface header fields

We added graphics allowing us to check how much a med is being dispatched each month, and we wanted to split it in two, so we ended having one bar graphic and a line one for the same data. So in a simple view we can check and analyze the differences without really needing to be interpreting the data.

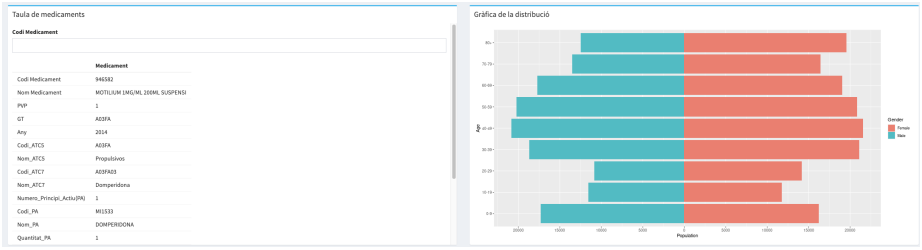


Figure 6: Med table and age graphic

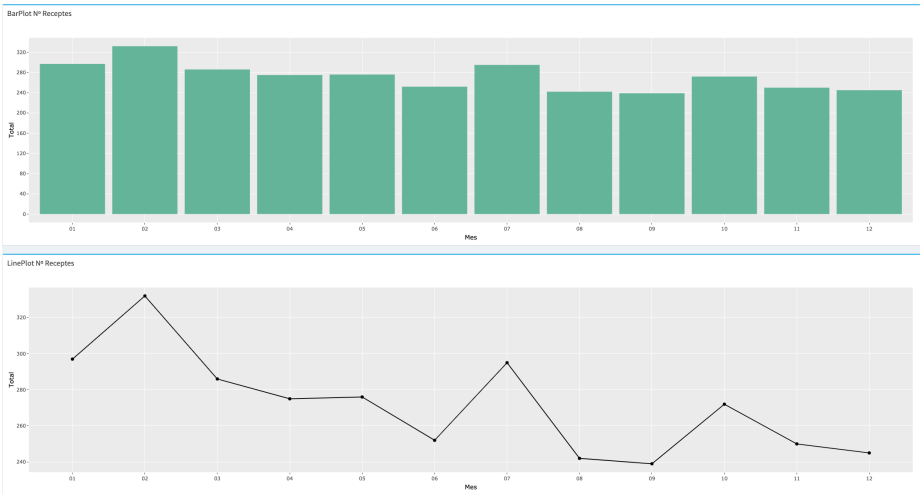


Figure 7: Bar and line graphic

Following the same idea of being able to show and check the data fast, we implemented a map of Lleida, so we were able to check how much a medicament was being receipted on each site. The map was moved outside the main page to a submenu, so it will only fetch as soon as we move to the submenu, improving the access time.

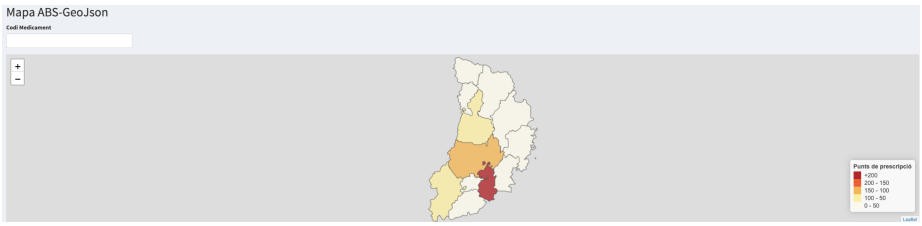
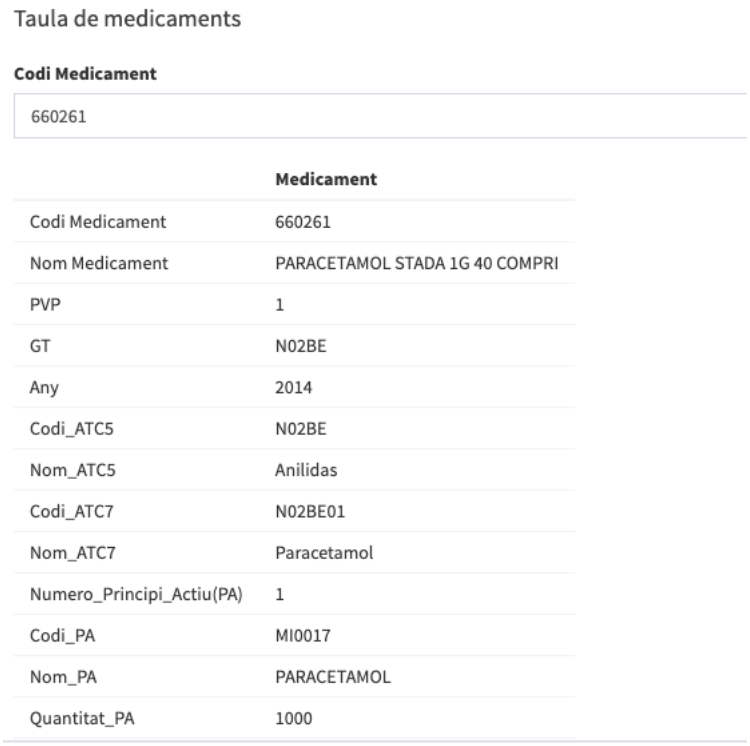


Figure 8: ABS Map

3 Use Case

As soon as we access to the dashboard, a preloaded med will be used to fill the table with all the info from it and both of the receipt graphics.

Using the text field in the top of the table, we can input the specific med we want to check.



Taula de medicaments

Codi Medicament

660261

	Medicament
Codi Medicament	660261
Nom Medicament	PARACETAMOL STADA 1G 40 COMPRI
PVP	1
GT	N02BE
Any	2014
Codi_ATC5	N02BE
Nom_ATC5	Anilidas
Codi_ATC7	N02BE01
Nom_ATC7	Paracetamol
Numero_Principi_Actiu(PA)	1
Codi_PA	MI0017
Nom_PA	PARACETAMOL
Quantitat_PA	1000

Figure 9: Use Case Step 1

At the same time the table will be filled with the new information, both graphs will be updated too. The values defined on the graphs will be automatically adapted from the new data that will be fetched.

Figure 10: Use Case Step 2

To be able to check the exact data that is being displayed we can only move the cursor on top of it and a legend will appear with the exact value and the column name.

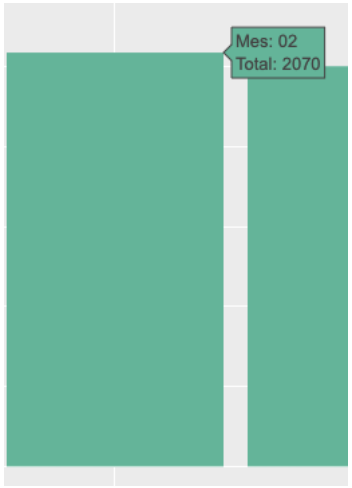


Figure 11: Use Case Step 3

Both receipts graphics contains different additional actions like enable pike lines, hide/show the different columns and more.

Figure 12: Use Case Step 4

On the side menu there will be the option to access to the map divided with the different ABS zones.

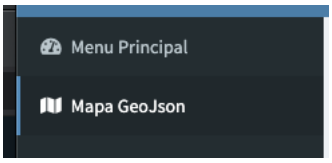


Figure 13: Use Case Step 5

Once again, data will be loaded to use the same predefined med mentioned in the previous menu.

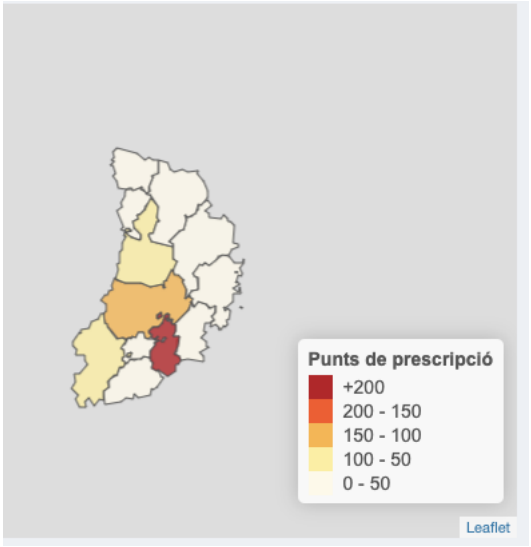


Figure 14: Use Case Step 6

Using the input on top of it allows as to load fetch the data from the specified med.

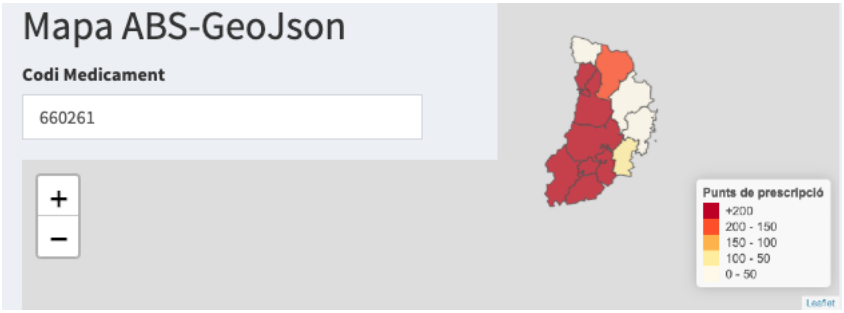


Figure 15: Use Case Step 7

4 Conclusions and Future Work

Working in a project that involved the health system data was a great experience, all the tools developed, and the environment set were done with the idea of allowing the professionals that could need to access or would like to use the data being able to use it easily or even allowing them to start from the development done and improving or adapting it to their needs.

Even if we still not have the sector feedback. The main objective and future of our project is publishing it to make medical data more accessible and trying to open new doors to the professionals to include it in future research. Making easiest as possible the interaction with the data and setting a base that could be improved and expanded.

One of our final objectives that could not be done due time, was to use our data and non-supervised algorithms to try to find different patterns that could appear.

The different developments made to fullfill that project are accessible at the GitHHub repository created for it: github.com/ricardPlana/TFG_RPlana

References

[1] Wullianallur Raghupathi and Viju Raghupathi. Big data analytics in healthcare: promise and potential. *Health Information Science and Systems*, 2(1):1–10, dec 2014.

[2] Microsot. Microsoft access, database software and applications. `https://www.microsoft.com/es-es/microsoft-365/excel`.

[3] Microsot. Microsoft excel. `https://www.microsoft.com/es-es/microsoft-365/excel`.

[4] DONALD FIRESMITH. Virtualization via containers. `https://insights.sei.cmu.edu/blog/virtualization-via-containers/`.

[5] Docker. What's a docker container. `https://www.docker.com/resources/what-container`.

[6] MongoDB. Execution of operations, max write batch size. `https://docs.mongodb.com/manual/reference/method/db.collection.insertMany/#execution-of-operations`.